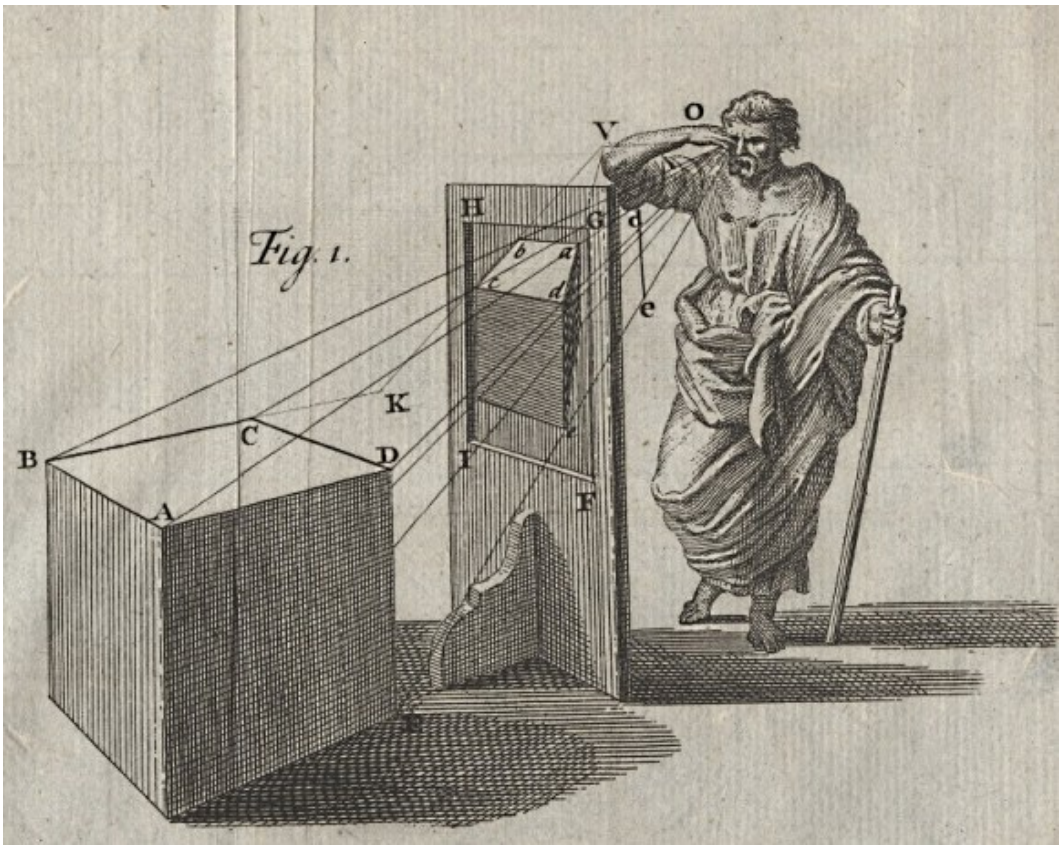
CSC 305 Lab 7
Preparation for a ray tracer

In this lab, we will go through the basic settings for implementing a ray tracer. In writing a simple ray tracer, we always prepare the whole image by hand, and then post it onto the screen, either by openGL or Qt drawing routines. This is different from the openGL-based drawing we have been practising until now. In previous labs, we pass many pieces of geometry to the openGL runtime, and the openGL runtime is in charge of maintain a single frame buffer onto which it will draw all our geometries. On the other hand, in the ray tracer lab we will practise creating an image from the most fundamental level: we render the scene pixel by pixel.



Brook Taylor, *New Principles of Linear Perspective,* 1749.
In this figure, the viewpoint is the eye (O); the HGFI plane corresponds to the image plane where we put our rendered image. Rays (such as OaA, ObB, etc.) are traced to project the cube face (ABCD) in the world space to the planar shape (abcd) in the image space.

Useful Qt routines

The Qt class we will be using extensively in this lab is the `QImage` class. As we noted above, in writing a ray tracer we need to construct the image pixel by pixel. A straightforward way of doing this is by using the
`QImage::setPixel(int x, int y, uint rgb)`
method. As its name implies, this method will set the pixel located at coordinate `(x, y)` into the color of `rgb`. Obviously, a routine calculate all pixel on a given rendering image would look like:

```
QImage myimage(renderWidth, renderHeight, QImage::Format_RGB32);

for (int i = 0; i < renderWidth; ++ i)
   for (int j = 0; j < renderHeight; ++ j)
   {
    //calculate pixel (i, j)'s R, G and B values by ray tracing
      myimage.setPixel(i, j, qRgb(R, G, B));
   }
```

This structure should suffice for most of your assignment requirements. It should be noted that the `setPixel(i, j, uint)` method is quite slow. If you are going for good performance, consider using `QImage::scanLine(int)`, and generate one row of pixels at a time. The other useful routines, such as open, save and display images with openGL, are also demonstrated in the code framework.

Ray-tracing a circle

We demonstrate the basic steps of ray tracing by the following example of drawing a circle. The procedure is similar to openGL-based rendering, except for that everything has to be done by hand. Here is a list of important steps:

1. Pick your viewpoint, which will be the origin (O) point for all your rays.
2. Establish your image plane (picture plane) in world space, and translate every pixel's coordinate onto their world space coordinates.
3. For each pixel, calculate its directional vector (D) by create a vector starting from the viewpint (O) to the pixel's world space coordinate. This will define the ray for the pixel as R = O + tD.
4. Calculate the ray-object intersections to figure out whether this ray hit any object in your scene.

We demonstrate this idea by the following simplified ray tracer, which will trace a circle parallel to the X-Y plane at Z = 10. Notice we didn't translate the pixel's coordinate from image space into world space in this example. We assume our image plane spans for the region (0, 0, 0) to (width, height, 0) in world space. This will make the world-space image plane too big for most applications. Please write appropriate coordinate transform for your assignment project.

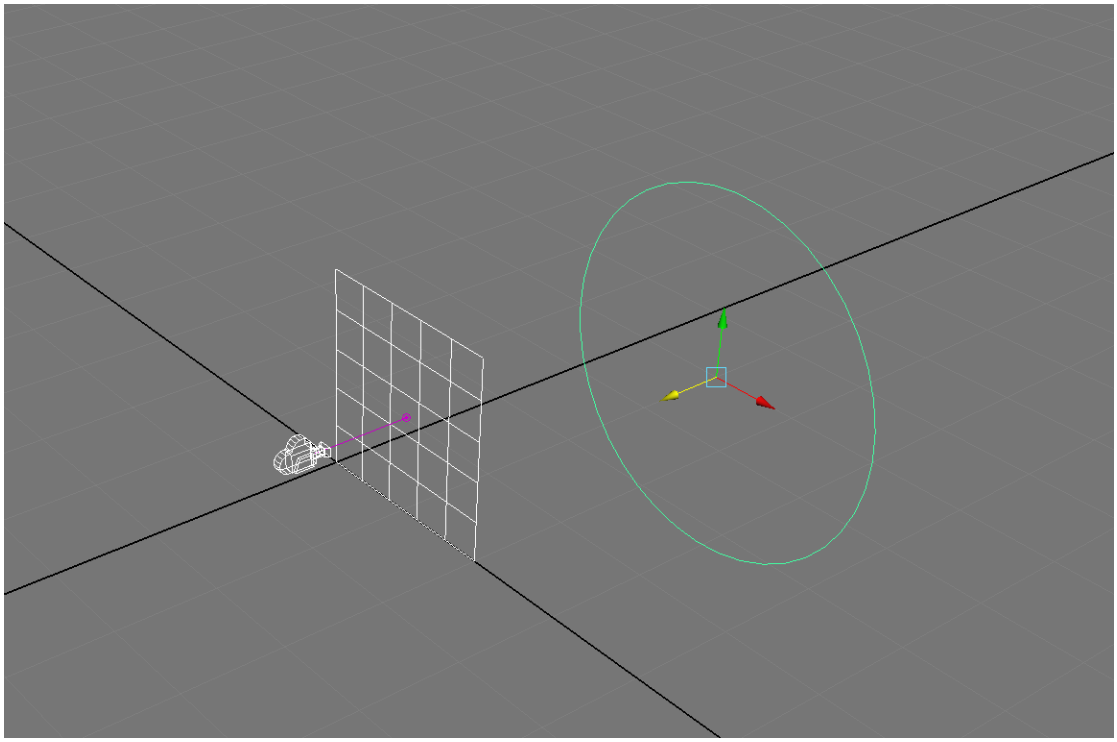


Figure 2. In our example, the image place is on the X-Y plane, and the circle is parallel to X-Y plane at Z = 10. Viewpoint is at the center of the image plane but offset to Z = -1. Code is listed in the next page.

```
    //This simple routine shows the elemetary ray arithmetics.
    //In your assignment, the more sophisticated ray tracer
should be implemented
    //in a separate class. See the tutorial document for details

    //Our image plane is at the Z = 0 plane.
    //The Camera is at the center of the image plane but offset
to negative Z Axis
    QVector3D CameraPoint(renderWidth / 2, renderHeight / 2, -
1);
    //We're going to ray-trace a circle at a positive Z location
    //parallel to the image plane.
    QVector3D CircleCenter( renderWidth / 2, renderHeight /2,
10);
    double CircleRadius = 400;


    for (int i = 0; i < renderWidth; ++ i)
        for (int j = 0; j < renderHeight; ++ j)
        {
            //Construct a ray
            QVector3D Origin = CameraPoint;
            QVector3D ImagePlanePosition(i, j, 0);
            QVector3D Direction = ImagePlanePosition - Origin;
            //Now we have a ray: R = O + t D
            //Solve for t at its intersection with the Z = 10
plane

            //The following math is specific to this example.
            //-- we know t = 11 in our scenario
            //Write your own solver for a real ray tracer

            QVector3D Z10Position = Origin + 11 * Direction;
            //See if the position is inside the circle
            QVector3D DistanceToCenter = Z10Position -
CircleCenter;
            double length = DistanceToCenter.length();
            //if it's inside, set it to white, otherwise set to
black
            if (length < CircleRadius)
            {

                myimage.setPixel(i, j, qRgb(255, 255, 255));
            } else
            {
                myimage.setPixel(i, j, qRgb(0, 0, 0));
            }
        }
```